2进程同步

2018年10月31日 13:37

-
- 进程同步

一. 进程同步的基本概念

1. 两种形式的制约关系
   1) **间接相互制约**关系：源于**互斥**访问临界资源
   2) **直接相互制约**关系：源于进程间的**合作**
2. 临界资源：一段时间内只允许一个进程访问的资源
3. 临界区(critical section)：在每个进程中访问临界资源的那段代码
   1) 保证诸进程互斥地进入自己的临界区即可实现对临界资源的互斥访问
   2) 在进入临界区之前，应先对欲访问的临界资源进行检查，如果此刻该临界资源正被某进程访问，则本进程不能进入临界区
   3) 检查临界资源的代码称为进入区(entry section)；恢复临界资源为未被访问的标志的代码称为退出区(exit section)；称进入区临界区退出区以外的代码为剩余区(remainder section)
4. 同步机制应遵循的规则
   1) **空闲让进**：无进程处于临界区时，临界资源处于空闲状态，应允许进程立即进入自己的临界区
   2) **忙则等待**：已有进程进入临界区时，表明临界资源正在被访问，其它进程必须等待
   3) **有限等待**：应保证进程在有限时间内能进入自己的临界区， 以免陷入“死等”状态
   4) **让权等待**：当进程不能进入自己的临界区时，应立即释放处理机，以免进程陷入“忙等”状态

二. 硬件同步机制

1) 用一个标志管理临界区，称其为锁

1. 关中断：进入锁测试前关闭中断，完成锁测试后打开中断
   1) 滥用关中断权力可能导致严重后果
   2) 关中断时间长，影响系统效率，限制处理器交叉执行程序的能力
   3) 不适用于多CPU系统，因为一个处理器上关中断并不能阻止其他处理器
2. Test-and-Set指令/原语
   1) boolean TS(boolean*lock){

```
boolean old=lock;
*lock=TRUE;//无论原值多少，调用了TS后总变为true
return old;}
```

   2) do{

```
while(TS(&lock))  ;
/*critical section*/
lock=FALSE;
/*remainder section*/
}while(TRUE);
```

3. swap指令，如Intel 80X86的XCHG指令
   1) do{

```
key=TRUE;
do{
    swap(&lock,&key);
}while(key!=FALSE);
}while(TRUE);
```

4. 上述都是忙等，不符合让权等待原则，浪费处理机，难于解决复杂同步问题

三. 信号量机制Semaphore

1. 整型信号量：一个用于标志资源数目的整形量S
   1) 除初始化外，仅能通过两个标准的原子操作wait(S)和 signal(S) 来访问
   2) 等待和信号量操作又称为 **P、V** 操作（荷兰文的通过和释放）

3) wait(S){
while (S<=0);
/*死循到S为正*/
S--; }
4) signal(S){
S++;}
5) 原子操作不可中断，因而保证了只有一个进程可修改信号量
6) 未遵循让权等待，会“忙等”
7) S<0时，**绝对值为阻塞进程数**

2. 记录型信号量：除了整型量记录资源数以外，还有链表指针记录阻塞队列
1) typedef struct s{
int value;
struct process_control_block*list;
}semaphore;
2) wait(semaphore*S){
S->value--;
if(S->value<0)
block(S->list);}
3) signal(semasphore*S){
S->value++;
if(S->value<=0)
wakeup(S->list);}
4) value<0，表示该类资源已分配完毕，进程应调用 block 原语，进行自我阻塞，放弃处理机，并插入到阻塞队列list中，做到让权等待
5) value<0时，其绝对值为阻塞队列中进程数
(1) 因而signal后若仍<=0时应调用wakeup原语
6) **互斥信号量：value初值为1**的信号量，其功能为实现互斥访问资源

3. AND型信号量：相当于同时对多个信号量做wait操作，用逻辑AND连接
1) 运行过程中需要的所有资源一次性全分配给进程，待使用完后一起释放
2) 只要尚有一个资源未能分配给进程，其它资源都不分配
3) Swait(S1,S2,…,Sn) {
while(TRUE){
if(S1>=1&&.....&&Sn>=1){
for(i=1;i<=n;i++) Si--;
break;
}else{
place the process in the waiting queue associated with the first Si found with Si<1，and set the program count of this process to the beginning of Swait operation}}}
Ssignal(S1,S2,…,Sn){
while(TRUE){
for(i=1;i<=n;i++){
Si++;
Remove all the process waiting in the queue associated with Si into the ready queue. }}}

4. 信号量集：先测试资源数是否大于下限值t，再减去需求值d个信号量
1) Swait(s1,t1,d1,…,Sn,tn,dn) {
while(TRUE){
if(S1>=t1&&.....&&Sn>=tn){
for(i=1;i<=n;i++) Si-=di;
break;
}else{
place the process in the waiting queue associated with the first Si found with Si<1，and set the program count of this process to the beginning of Swait operation}}}

```
Ssignal(S1,d1,...,Sn,dn){
    while(TRUE){
        for(i=1;i<=n;i++){
            Si+=di;
            Remove all the process waiting in the queue associated with Si into the ready queue. }}}
```

2) Swait(S,d,d)只有一个信号量S，允许它每次申请d个资源
3) Swait(S,1,1)蜕化为一般的记录型信号量(S>1时)或互斥信号量(S=1时)
4) Swait(S,1,0)相当于一个可控开关，只起检查作用。当S≥1时，允许多个进程进入某特定区；当S变为0后，将阻止任何进程进入特定区

四. 信号量的应用

1. 信号量实现进程互斥：为临界资源设置一互斥信号量mutex，初值为1，将各进程访问该资源的临界区CS置于wait(mutex)后signal(mutex)前
    1) mutex取1、0、-1对应都未进入临界区、一个进入、一个进一个阻
    2) wait和signal必须成对出现，wait实现互斥、阻塞，signal实现释放临界资源、唤醒被阻进程
2. 信号量实现前趋关系： P1和P2共享一个公用信号量S，初值为0，在S1后signal(S)；在S2前wait(S)
3. 信号量实现进程同步：s1=0,s2=0;

    写 signal(s1) wait(s2) 循环
    wait(s1) 读 signal(s2) 循环
    或s2初值取1，让即可让写进程和读进程格式一样

五. 管程机制

1. 管程monitor：由共享资源的数据结构，以及由对该数据结构实施操作的一组过程所组成的资源管理模块
    1) 管程的组成：管程名、数据结构说明、对该数据结构进行操作的过程、对共享数据设初值的语句
    2) 封装于管程内的数据仅能被封装于管程内的过程访问，反之亦然
    3) 一次只准许一个进程进入管程，实现了互斥
    4) 特性：
        (1) 模块化，它是基本程序单位，可单独编译
        (2) 抽象：既有数据又有操作
        (3) 信息掩蔽：数据结构和过程实现对外不可见
    5) 管程和进程：
        (1) 进程中的数据结构是私有PCB；管程的是公有数据结构，如数据队列
        (2) 进程由顺序执行有关操作；管程只进行同步和初始化
        (3) 进程实现并发性；管程解决互斥问题
        (4) 进程可像调用子程序一样调用管程；管程只能被动工作
        (5) 进程能并发；管程不能与调用者并发
        (6) 进程有动态性，由创建/撤销而诞生/消亡；管程只是系统中一个供调用的资源管理模块
2. 条件变量
    1) 当一个进程调用了管程，在管程中时被阻塞或挂起，但不释放管程， 则其它进程无法进入管程，被迫长时间地等待。为了解决这个问题，引入了条件变量condition
    2) 在管程中对多个阻塞/挂起原因设置了多个条件变量，对这些条件变量的访问，只能在管程中进行
    3) 条件变量也是一种抽象数据类型，对条件变量的操作仅仅是wait和signal，每个条件变量保存了一个链表，用于记录因该条件变量而阻塞的所有进程
        (1) x.wait：正在调用管程的进程因x条件需要被阻塞或挂起，则调用x.wait将自己插 入到x条件的等待队列上，并释放管程，直到x条件变化。此时其它进程可以使用该管程。
        (2) x.signal：正在调用管程的进程发现x条件发生了变化，则调用x.signal，重新启动一个因x条件而阻塞或挂起的进程，如果没有，则继续执行原进程，而不产生任何结果
            i. 信号量机制中的signal操作总是要执行s:=s+1操作，因而总会改变信号量的状态

    P唤醒Q后，确定哪个执行，哪个等待，可采用下述两种方式之一进行处理：P等待，直至Q离开管程或等待另一条件/Q等待，直至P离开管程或等待另一条件

Hoare采用了第一种处理方式，而Hansan选择了两者的折衷，他规定管程中的过程所执行的signal操作是过程体的最后一个操作，于是， 进程P执行signal操作后立即退出管程，因而进程Q马上被恢复执行

◆

- 经典进程的同步问题

一. 生产者-消费者问题(The proceducer-consumer problem)

1. 记录型信号量

1) mutex实现互斥进入缓冲池，empty、full实现判断能否继续生产/消费

```
Semaphore mutex=1,empty=n,full=0;
int in=0,out=0;
item buffer[n];
void producer(){
    while(TRUE){
        /*produce the item :nextp*/
        wait(empty);//等到不空时
            wait(mutex);//等到可以进入池时
                buffer[in]=nextp;
                in=(in+1)%n;
            signal(mutex);
        signal(full);}}
void consumer(){
    while(TRUE){
        wait(full);
            wait(mutex);
                nextc=buffer[out];
                out=(out+1)%n;
            signal(mutex);
        signal(empty);
        /*consume the item :nextc*/}}
void main(){
    cobegin
        producer();  consumer();
    coend }
```

2) 两个信号量的wait写反了可能引起死锁

2. AND型信号量：将相邻两个wait/signal合并成一个Swait/Ssignal即可
3. 管程：为缓冲池建立管程，包含put和get过程实现放/取产品；将空/满信号量作为两个条件变量，对应cwait和csignal过程，和一个阻塞队列

二. 哲学家进餐问题

1. 记录型信号量

```
semaphore chopstick[5]={1,1,1,1,1};
while(TRUE){
    wait(chopstick[i]);
    wait(chopstick[(i+1)%5]);
        /*eat*/
    signal(chopstick[i]);
    signal(chopstick[(i+1)%5]);
        /*think*/}
```

1) 可能导致死锁，解决方法：
   (1) 至多允许同时四人拿筷子
   (2) 规定部分人先拿左，部分人先拿右
   (3) 同时检查左右可用，才可以拿，类似下一条

2. AND信号量：两个信号量一起Swait/Ssignal即可，不会死锁

三. 读者-写者问题

1. 记录型信号量

1) 写需要互斥信号量，读可以多进程同时读

2) 无读者在读时才可能在被写，才需要等待写操作/通知写操作
3) ↑读者数应是一个临界资源，需要为其再设置一个互斥信号量
4) 读者数是整型变量，不断++，非零时写者需要等

```
semaphore wmutex=1,rmutex=1;
int readercount=0;
void reader(){
    while(TRUE){
        wait(rmutex);
            if(readercount==0)
                wait(wmutex);
            readercount++;
        signal(rmutex);
        /*read*/
        wait(rmutex);
            readercount--;
            if(readercount==0)
                signal(wmutex);
        signal(rmutex);}}
void writer(){
    while(TRUE){
        wait(wmutex);
            /*write*/
        signal(wmutex);}}
```

2. 信号量集
   1) 可限制读者数最多RN，通过一个初值为RN的信号量不断wait(L,1,1)
   2) Swait(wmutex,1,0)起开关作用，只是在读之前检查是不是在写
   3) 读者数是信号量，不断--，非初值时写者需要等

```
int RN;
semaphore wmutex=1,rMax=RN;
int readercount=0;
void reader(){
    while(TRUE){
        Swait(rMax,1,1);
            Swait(wmutex,1,0);
            /*read*/
        Ssignal(rMax,1);}}
void writer(){
    while(TRUE){
        Swait(wmutex,1,1 , rMax,RN,0);
            /*write*/
        Ssignal(wmutex,1);}}
```

i.
ii.
iii.
iv.
v.
vi.
vii.
viii.
ix. --------我是底线----------